System- / Objektdesign
mario world wars

Projekt: **mario world wars**
Voraussetzung: Pflichtenheft
Autoren: *Thomas Vogel, Michael Palmer, Florian Oeser*
Kontakt: **mario@link2wall.de**
Letzte Änderung: **14.10.08**

Inhaltsverzeichnis

1 Systemdesign....4
1.1 Hardware....4
1.2 Datenhaltung....4
1.3 Netzwerkkommunikation....6

2 Objektdesign (Enginearchitektur)....7

3 GUI-Design....13
3.1 Hauptmenü....13
3.2 Einstellungen....14
3.3 Steuerungseinstellungen....15
3.4 Grafikeinstellungen....16
3.5 Nachrichtenbox....17
3.6 Charakterdesign....18

1 Systemdesign

Das Spiel „Mario World Wars" ist eine **Standalone**-Applikation. Die Software wird auf dem Betriebssystem Windows installierbar sein und bringt in einem Setup alle anderen Abhängigkeiten, wie etwa die XNA Runtimes, mit.

1.1 Hardware

Zur **Hardware** des Anwender-Computers sind folgende Voraussetzungen gegeben.

CPU: > 1Ghz bei einem FSB ab 533 Mhz

GPU: >= 256 MB Ram

RAM: >= 1 Gbyte

1.2 Datenhaltung

Zur **Datenhaltung** der Spieler-Informationen nutzen wir eine vom Programm generierte XML-Datei, die über Informationen verfügt, welche wir über einen Spieler halten möchten.

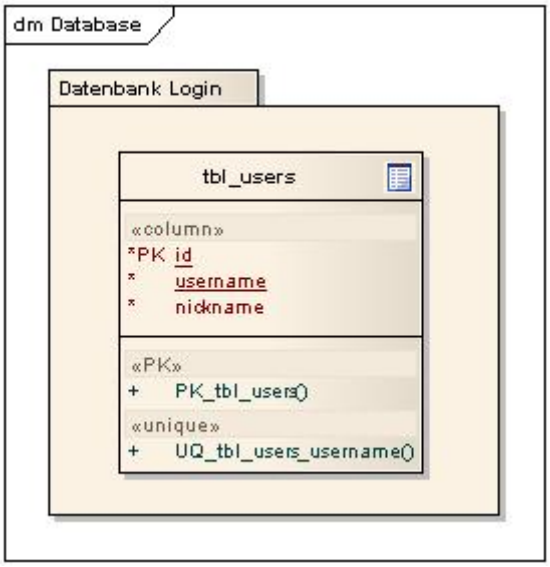
XML: Spielername, Tastaturbelegung, Geräuscheinstellungen, Grafikeinstellungen

So könnte diese Datei aussehen:

```
<playerwood>
	<player>
		<name>supermario</name>
		<keyboard>
			<jump>x</jump>
			<forward>up</forward>
			<backward>down</backward>
			<left>left</left>
			<right>right</right>
			<spezialweapon>y</spezialweapon>
		</keyboard>
		<sounds>
			<musik-volume>10</musik-volume>
			<game-volume>50</ame-volume>
		</sounds>
		<graphics>
			<resolution>1024x786</resolution>
			<colors>32</colors>
			<anti-alias>on</anti-alias>
			<widescreen>on</widescreen>
		</graphics>
	</player>
</playerwood>
```

Des weiteren haben wir eine Datenbank in der wir die Benutzer speichern. Jeder Benutzer muss sich, bevor er spielen kann, bei uns registrieren. Die Registrierung wird bei jedem Login abgefragt. Die notwendigen Datenbankoperationen setzen wir mit C# und dem .NET-Connector für MySQL um.

In der Datenbank haben wir lediglich eine Tabelle:



In der Tabelle haben wir logischerweise einen Primärschlüssel und zwei weitere Felder. Zum einem einen eindeutigen Benutzername und einen Nickname der im Spiel verwendet wird.

1.3 Netzwerkkommunikation

Diese Applikation wird ein Standalone-Spiel sein, welches über Netzwerk kommunizieren kann, um dort gegen andere Spieler anzutreten. Um dies zu realisieren wird eine Socket Verbindung hergestellt. Das Protokoll welchem dies zu Grunde liegt, nennt sich UDP (User Datagramm Protocoll).

UDP Ist ein verbindungsloses Netzwerkprotokoll welches meist für „realtime" Spiele wie dieses genutzt wird. Das TCP (Transmission Control Protocoll) welches verbindungsorientiert arbeit, eignet sich nur bedingt für solche Spiele da hier das Zustellen der Pakete durch eine „three-way-handshake" gewährleistet wird und damit die Kommunikation über das Protokoll verlangsamt. Dies kann/wird zu Performanceverlust führen, welcher bei Spielen, wo die Geschwindigkeit entscheidet, von besonderer Relevanz ist.

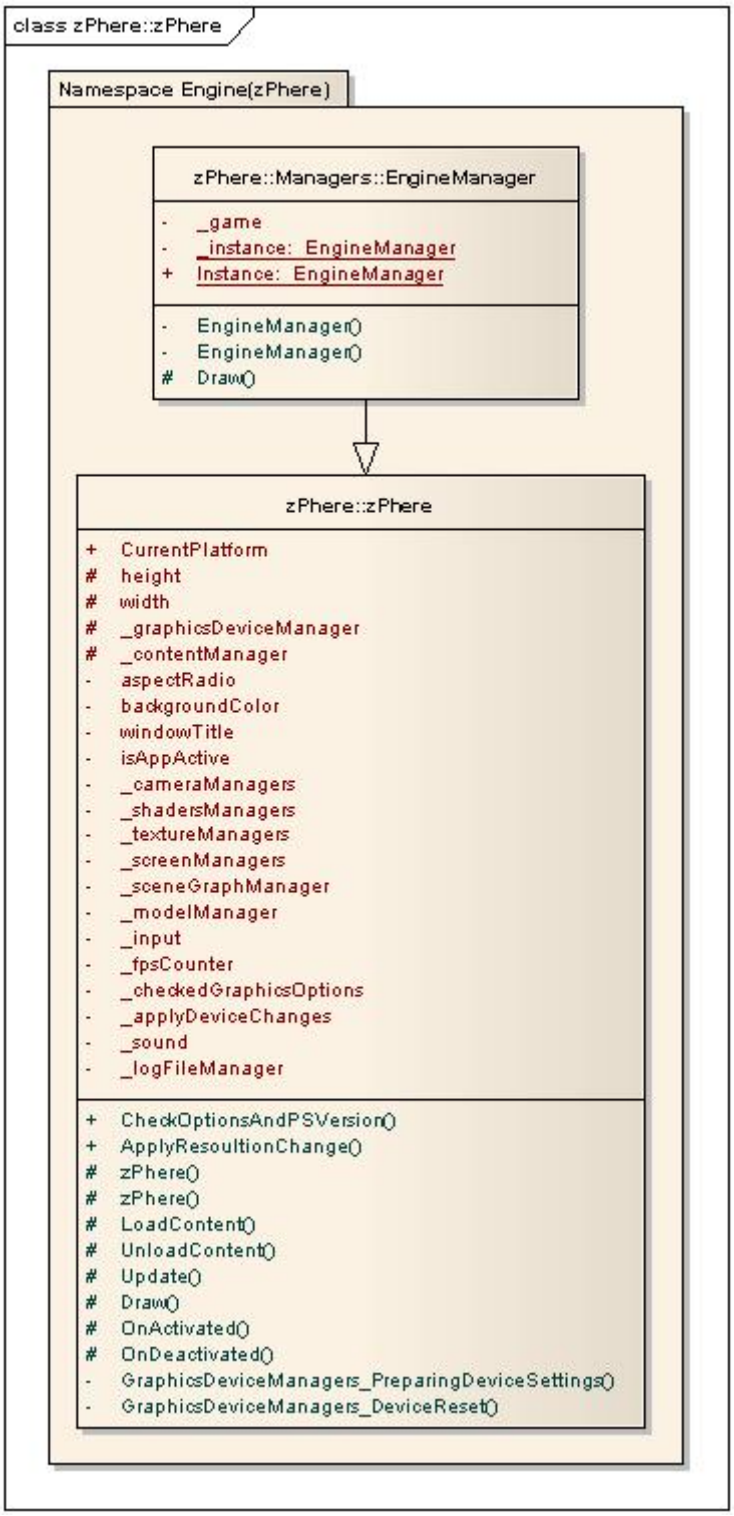
Jedoch werden wir die Anmeldung des Spielers an unserem Server(Login) aus Sicherheitsgründen mit TCP umsetzen.

Host und Client:

Bei Spieleröffnung wird ein Spieler zum so genannten Host. Dieser stellt die Schnittstelle der Daten zur Verfügung. Weitere Spieler (in diesem Fall bis zu sieben Clients) verbinden sich dann mit der IP-Adresse des Host, um ihre Daten (Position) an diesen Server zu senden und die Daten (Positionen) der anderen Gegenspieler zu erfahren. Dazu wird auf den gleichen Port zugegriffen, welcher die Brücke zwischen der Applikation und dem Protokoll herstellt. Dieser Port wird sich im Bereich zwischen 5000 und 65535 bewegen.

2 Objektdesign (Enginearchitektur)

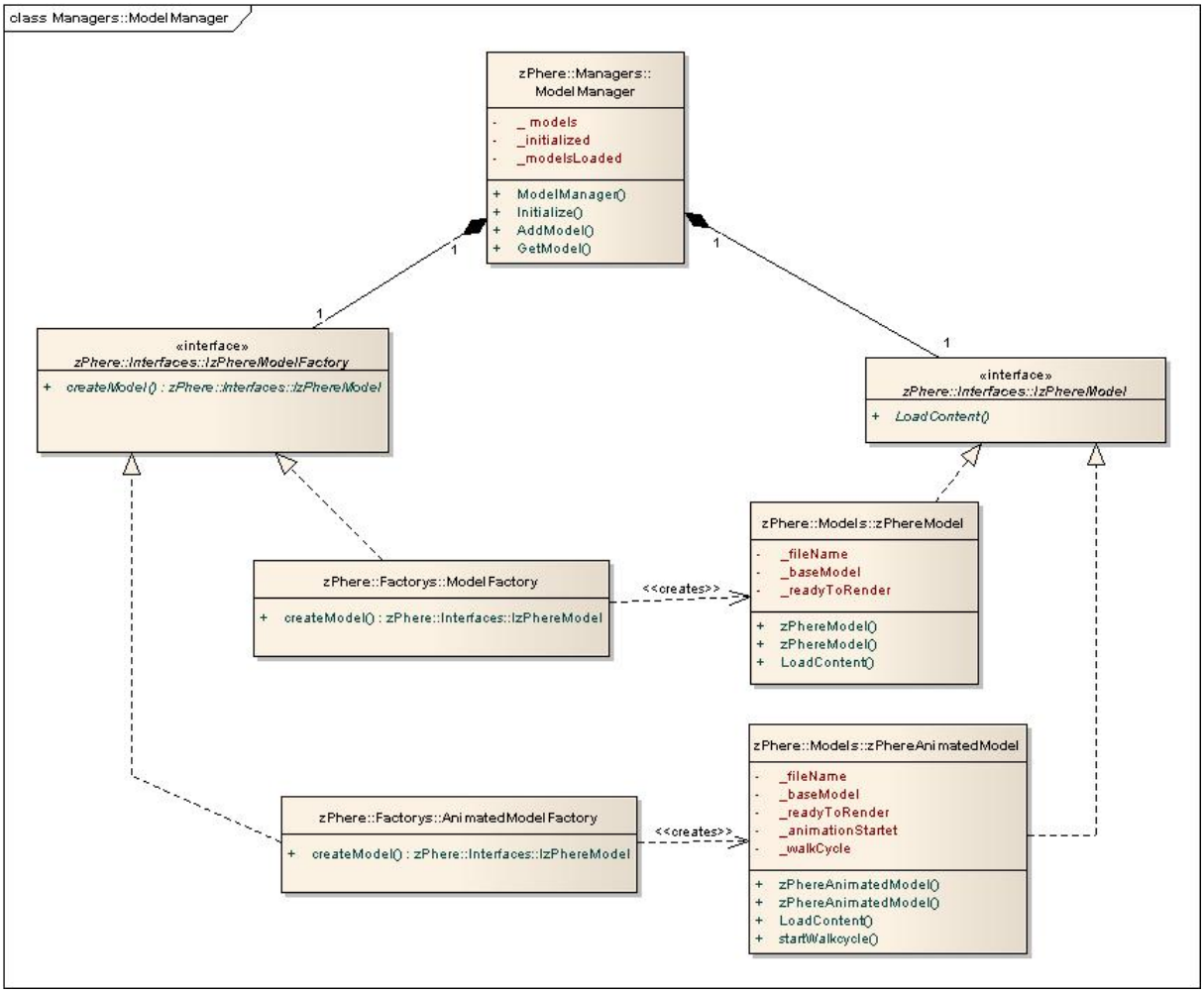
Die Engine bzw. der EngineManager wird als Singleton implementiert. Das heißt es ist lediglich eine Instanz des Managers möglich. Es ist ein allgemein übliches Prinzip den Enginekern so vor Mehrfachinstanzierung zu schützen.



Auch die anderen Manager(Scenegraph, Kamera etc.) könnte man so umsetzten, da

diese ebenfalls nur eine Instanz benötigen.

Weiterhin verwenden wir das Abstract Factory Pattern für die Umsetzung verschiedener Models, Shader und Texturen. Der Vorteil des Patterns ist, das der Client(in unserem Falle der jeweilige Manager) nicht die konkrete Implementierung der Klassen kennen muss(OO-Prinzip der Kapselung). Er arbeitet lediglich mit einem Interface und bekommt von der Factory das entsprechende konkrete Objekt, welches durch diese Factory selbst erstellt wurde, zurück.



Wir haben lediglich den Aufbau für den Modelmanager modelliert. Wie erwähnt läuft die Implementierung analog für den Shader- und Textur Manager. Dort werden dann verschiedene Shader- und Texturtypen implementiert. Es ist in Planung die Factory's und Interfaces in separate Bibliotheken(DLL's) auszulagern und somit weiter vor der Engine(ModelManager etc.) zu kapseln.

Das würde außerdem den Vorteil mit sich bringen das auszuliefernde Updates, bei Änderungen in den konkreten Klassen, kleiner wären, da der entsprechende Teil nur ausgetauscht werden müsste. Logischerweise nur wenn die Interfaces sich nicht ändern.

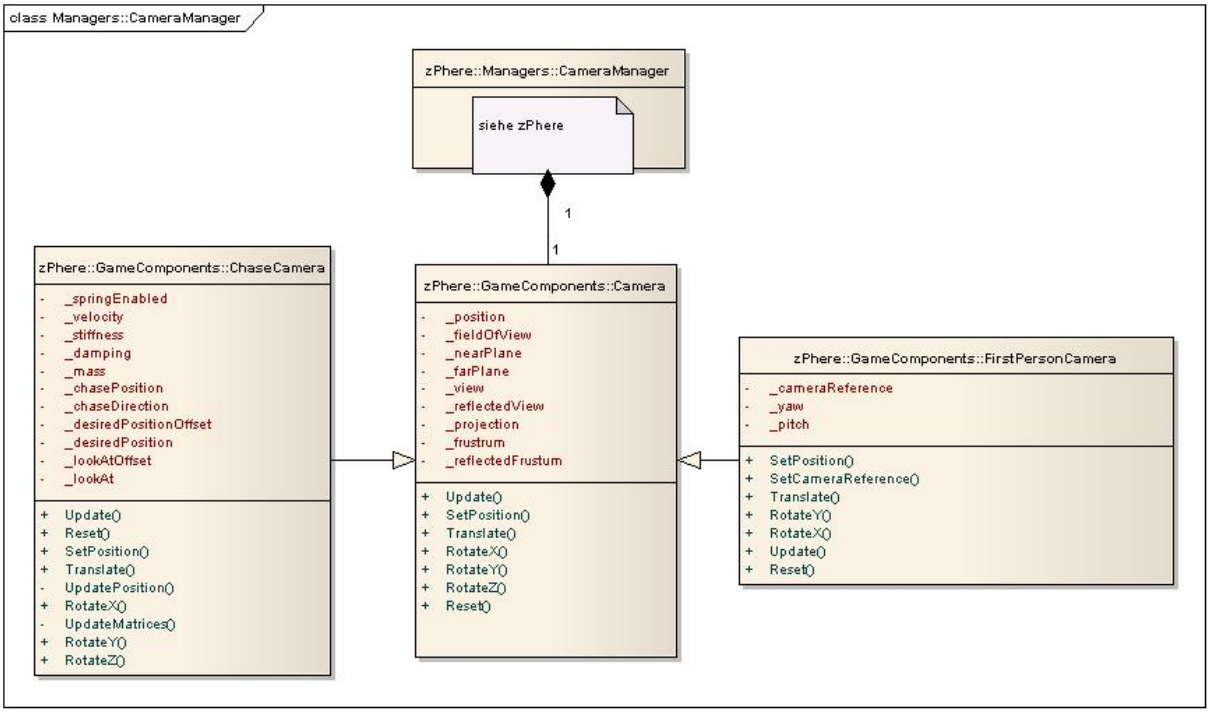
Weiterhin werden wir ebenfalls alle GameComponents wie Input, FPSCounter in separte Bibliotheken(auch DLL) auslagern. Das steht aber im Moment noch nicht fest, da wir die

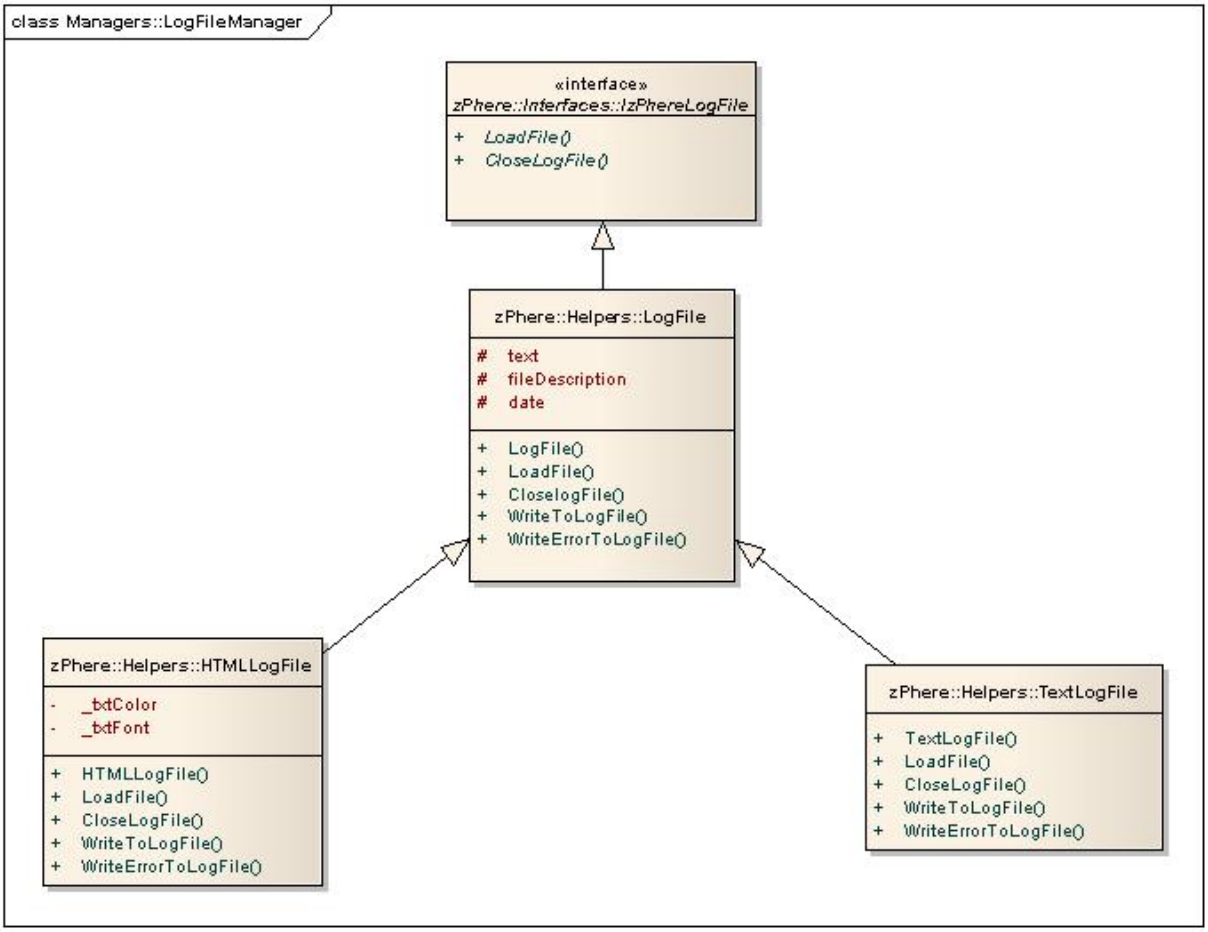
verschieden Abhängigkeiten noch nicht hinreichend prüfen können. Ein Vorteil wäre aber das Komponenten, die nicht benötigt werden, gar nicht erst geladen werden. Das ist möglich da DLL's nur zur Laufzeit geladen werden können. Dieser Ansatz folgt weiterhin der Implementierung einer üblichen Enginearchitektur, da sie komplett unabhängig von einem konkreten Spiel sein muss und es auch deshalb wichtig sein kann verschiedene Komponenten nicht von von herein zu benutzen.

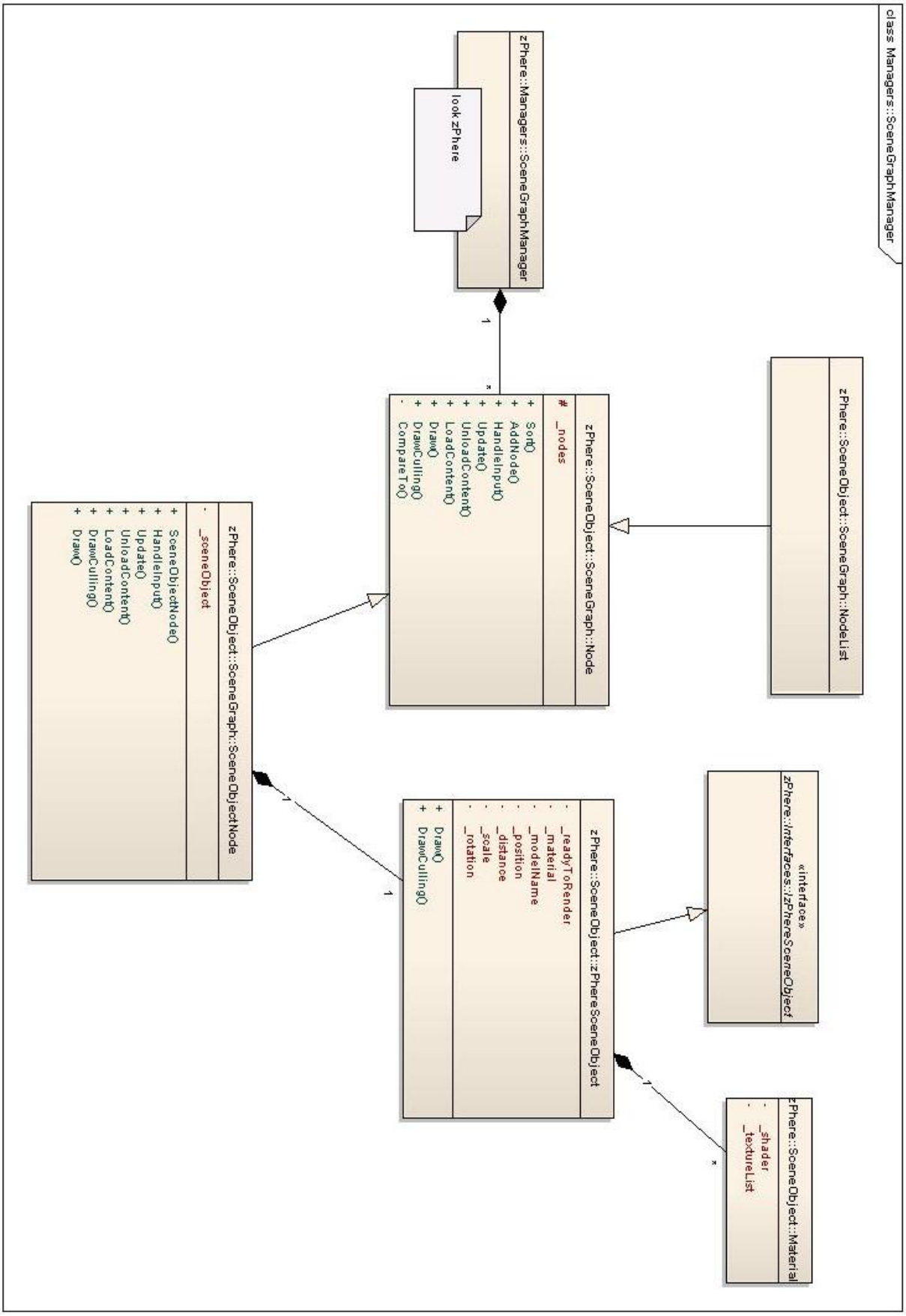
Es ist weiterhin in Planung alle Events über ein Observer Pattern zu kapseln.

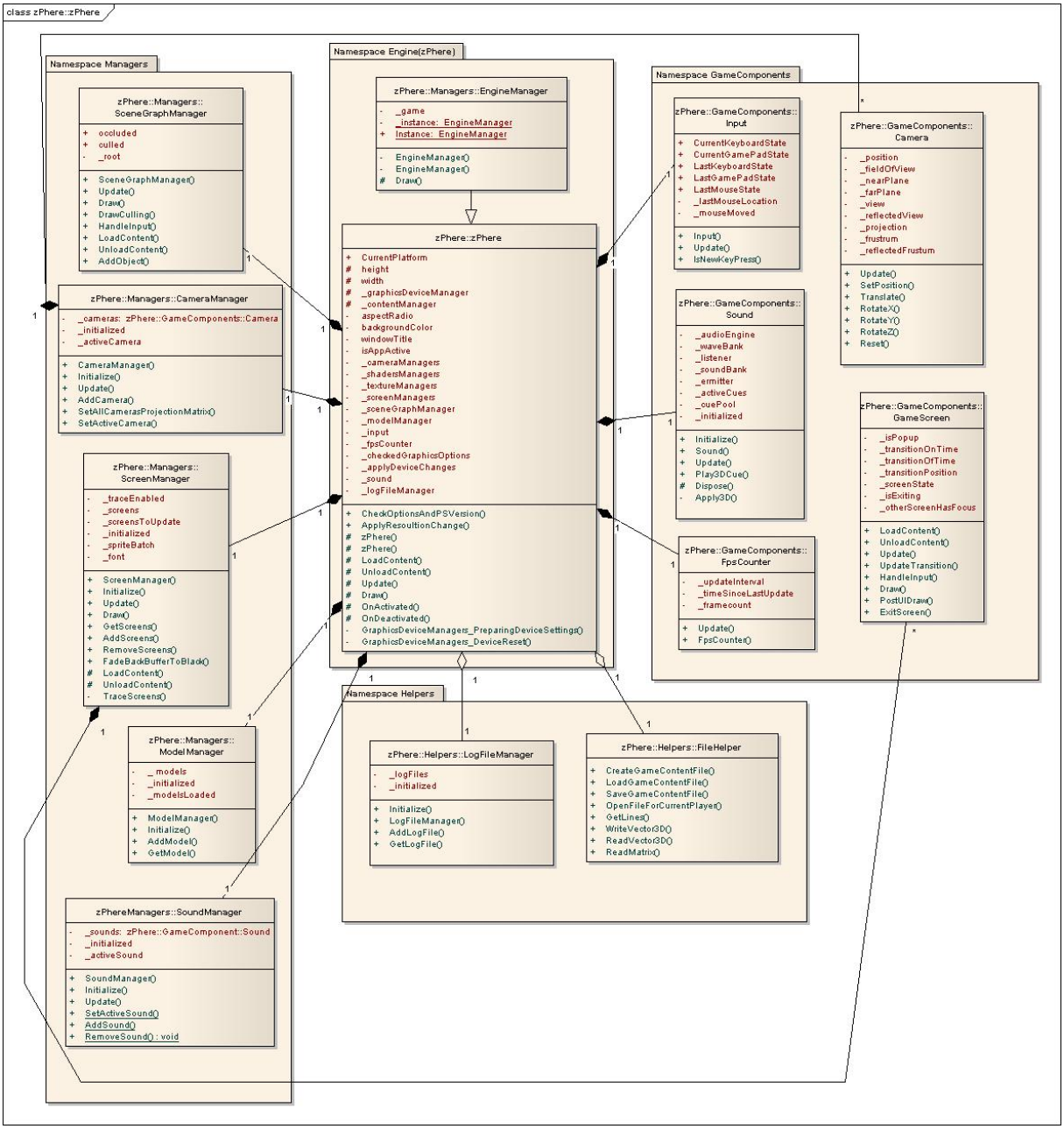
Ausserdem werden wir alle Texte(Strings) externalisieren um später bei der Lokalisierung besser arbeiten zu können. Dies könnte zum Beispiel über XML-Files geschehen.

Alle weiteren implementierungsspezifischen Aspekte hatten wir ja bereits im Anforderungsmodell dargestellt, wollen diese aber trotzdem nochmal aufführen.




class Managers::LogFileManager
«interface»
zPhere::Interfaces::IzPhereLogFile
+ LoadFile()
+ CloseLogFile()
zPhere::Helpers::LogFile
text
fileDescription
date
+ LogFile()
+ LoadFile()
+ CloselogFile()
+ WriteToLogFile()
+ WriteErrorToLogFile()
zPhere::Helpers::HTMLLogFile
- _txtColor
- _txtFont
+ HTMLLogFile()
+ LoadFile()
+ CloseLogFile()
+ WriteToLogFile()
+ WriteErrorToLogFile()
zPhere::Helpers::TextLogFile
+ TextLogFile()
+ LoadFile()
+ CloseLogFile()
+ WriteToLogFile()
+ WriteErrorToLogFile()


class Managers::SceneGraphManager
zPhere::Managers::SceneGraphManager
look zPhere
1
*
zPhere::SceneObject::SceneGraph::NodeList
zPhere::SceneObject::SceneGraph::Node
_nodes
+ Sort()
+ AddNode()
+ HandleInput()
+ Update()
+ UnloadContent()
+ LoadContent()
+ Draw()
+ DrawCulling()
- CompareTo()
zPhere::SceneObject::SceneGraph::SceneObjectNode
- _sceneObject
+ SceneObjectNode()
+ HandleInput()
+ Update()
+ UnloadContent()
+ LoadContent()
+ DrawCulling()
+ Draw()
«interface»
zPhere::Interfaces::IzPhereSceneObject
zPhere::SceneObject::zPhereSceneObject
- _readyToRender
- _material
- _modelName
- _position
- _distance
- _scale
- _rotation
+ Draw()
+ DrawCulling()
zPhere::SceneObject::Material
- _shader
- _textureList


class zPhere::zPhere
Namespace Managers
zPhere::Managers::
SceneGraphManager
+ occluded
+ culled
- _root
+ SceneGraphManager()
+ Update()
+ Draw()
+ DrawCulling()
+ HandleInput()
+ LoadContent()
+ UnloadContent()
+ AddObject()
zPhere::Managers::CameraManager
- _cameras: zPhere::GameComponents::Camera
- _initialized
- _activeCamera
+ CameraManager()
+ Initialize()
+ Update()
+ AddCamera()
+ SetAllCamerasProjectionMatrix()
+ SetActiveCamera()
zPhere::Managers::
ScreenManager
- _traceEnabled
- _screens
- _screensToUpdate
- _initialized
- _spriteBatch
- _font
+ ScreenManager()
+ Initialize()
+ Update()
+ Draw()
+ GetScreens()
+ AddScreens()
+ RemoveScreens()
+ FadeBackBufferToBlack()
# LoadContent()
# UnloadContent()
- TraceScreens()
zPhere::Managers::
ModelManager
- _models
- _initialized
- _modelsLoaded
+ ModelManager()
+ Initialize()
+ AddModel()
+ GetModel()
zPhereManagers::SoundManager
- _sounds: zPhere::GameComponent::Sound
- _initialized
- _activeSound
+ SoundManager()
+ Initialize()
+ Update()
+ SetActiveSound()
+ AddSound()
+ RemoveSound() : void
Namespace Engine(zPhere)
zPhere::Managers::EngineManager
- _game
- _instance: EngineManager
+ Instance: EngineManager
- EngineManager()
- EngineManager()
Draw()
zPhere::zPhere
+ CurrentPlatform
# height
# width
# _graphicsDeviceManager
# _contentManager
- aspectRadio
- backgroundColor
- windowTitle
- isAppActive
- _cameraManagers
- _shadersManagers
- _textureManagers
- _screenManagers
- _sceneGraphManager
- _modelManager
- _input
- _fpsCounter
- _checkedGraphicsOptions
- _applyDeviceChanges
- _sound
- _logFileManager
+ CheckOptionsAndPSVersion()
+ ApplyResoultionChange()
# zPhere()
# zPhere()
# LoadContent()
# UnloadContent()
# Update()
# Draw()
# OnActivated()
# OnDeactivated()
- GraphicsDeviceManagers_PreparingDeviceSettings()
- GraphicsDeviceManagers_DeviceReset()
Namespace Helpers
zPhere::Helpers::LogFileManager
- _logFiles
- _initialized
+ Initialize()
+ LogFileManager()
+ AddLogFile()
+ GetLogFile()
zPhere::Helpers::FileHelper
+ CreateGameContentFile()
+ LoadGameContentFile()
+ SaveGameContentFile()
+ OpenFileForCurrentPlayer()
+ GetLines()
+ WriteVector3D()
+ ReadVector3D()
+ ReadMatrix()
Namespace GameComponents
zPhere::GameComponents::
Input
+ CurrentKeyboardState
+ CurrentGamePadState
+ LastKeyboardState
+ LastGamePadState
+ LastMouseState
- _lastMouseLocation
- _mouseMoved
+ Input()
+ Update()
+ IsNewKeyPress()
zPhere::GameComponents::
Camera
- _position
- _fieldOfView
- _nearPlane
- _farPlane
- _view
- _reflectedView
- _projection
- _frustrum
- _reflectedFrustum
+ Update()
+ SetPosition()
+ Translate()
+ RotateX()
+ RotateY()
+ RotateZ()
+ Reset()
zPhere::GameComponents::
Sound
- _audioEngine
- _waveBank
- _listener
- _soundBank
- _emitter
- _activeCues
- _cuePool
- _initialized
+ Initialize()
+ Sound()
+ Update()
+ Play3DCue()
# Dispose()
- Apply3D()
zPhere::GameComponents::
FpsCounter
- _updateInterval
- _timeSinceLastUpdate
- _framecount
+ Update()
+ FpsCounter()
zPhere::GameComponents::
GameScreen
- _isPopup
- _transitionOnTime
- _transitionOfTime
- _transitionPosition
- _screenState
- _isExiting
- _otherScreenHasFocus
+ LoadContent()
+ UnloadContent()
+ Update()
+ UpdateTransition()
+ HandleInput()
+ Draw()
+ PostUIDraw()
+ ExitScreen()

3 GUI-Design

Die nun folgenden Screenshots der Spiel-GUI zeigen noch nicht das fertige und endgültige Aussehen, geben aber schon mal eine grobe Richtung vor. Insbesondere könnte sich die Typografie und das Farbschemata noch ändern.

3.1 Hauptmenü

Dieser Screenshot zeigt das Hauptmenü.

Bei Programmstart wird dieses Menü initialisiert. Hier stehen die wesentlichen Einstellungs-Möglichkeiten zur Verfügung. Über Beenden wird das Spiel verlassen und aus dem Speicher entfernt. Bei Auswahl der Credits erhält der Benutzer alle Hintergrundinformationen zum Spiel und den Entwicklern. Die weiteren Menüpunkte werden in folgenden Screenshots erklärt.

3.2 Einstellungen

Dieser Screenshot zeigt das Obermenü zu den Einstellungen die vorgenommen werden können. Mit 'Zurück' kommt man wieder in das Hauptmenü.

Steuerungseinstellungen und Grafikeinstellungen werden im System gespeichert und während des gesamten Spielverlaufs beibehalten.

3.3 Steuerungseinstellungen

Dieser Screenshot zeigt das Menü in dem Einstellungen bezüglich der Steuerung vorgenommen werden. Wir haben hier lediglich zwei beispielhafte Einträge, da wir für den Prototyp noch keine freie Steuerungswahl planen und somit auch noch nicht das entsprechende Menü festlegen können. Der Eintrag mit den drei Punkten soll dies noch einmal verdeutlichen.

Bei 'Speichern und zurueck' werden die vorgenommen Einstellungen in dem vorher ausgewählten Benutzerprofil gespeichert, so dass bei Neustart der Anwendung diese automatisch geladen werden.

3.4 Grafikeinstellungen

Dieser Screenshot zeigt das Menü indem Einstellungen bezüglich der Grafik vorgenommen werden.

Auswahlmöglichkeiten werden wahrscheinlich:

Auflösung → Angabe in Pixeln (z. B. 1024x768)

Anti-Aliasing → Generierung von weichen Übergängen zwischen den Pixeln

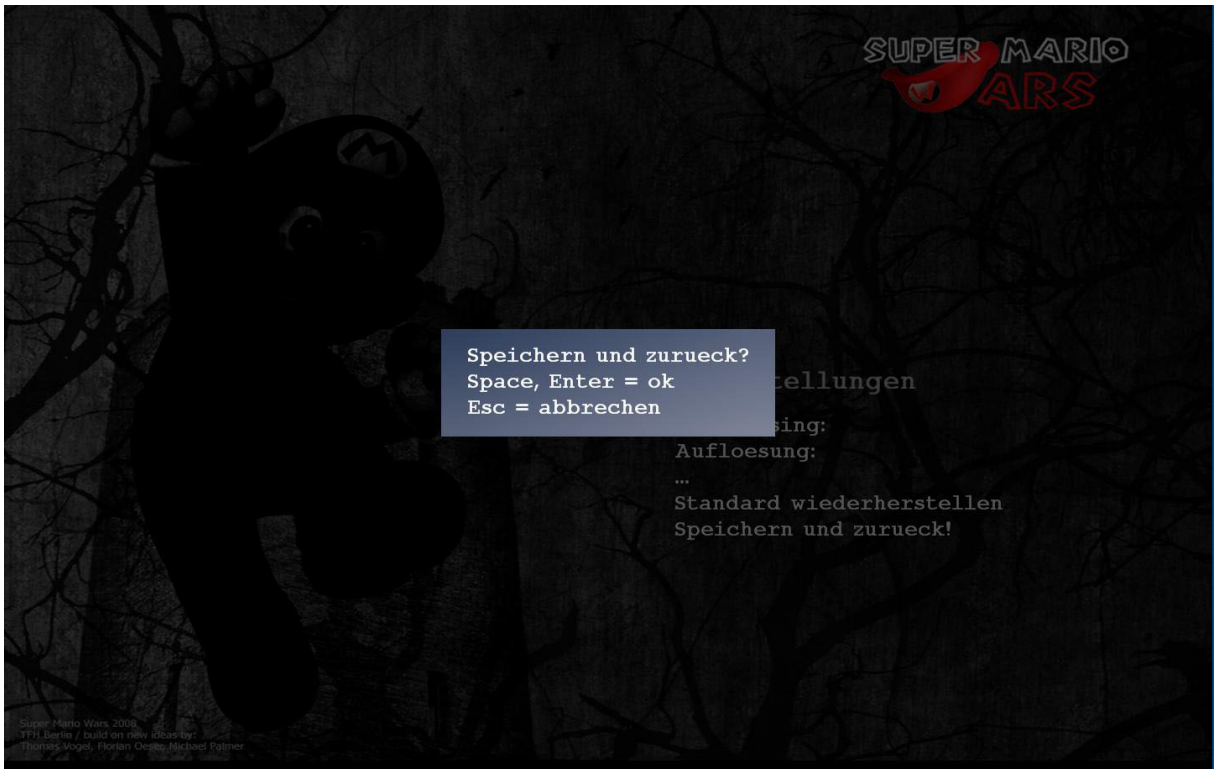
Farbtiefe → Angabe in Bit (z. B. 32)

Widescreen-Modus → Für Breitbildmonitore

3.5 Nachrichtenbox

Dieser Screenshot zeigt einen Entwurf unserer Message-Box. Diese verwenden wir in den beiden Einstellungsmenüs. Es sind Nachrichtenboxen oder Meldefenster welche den Benutzer fragen ob er wirklich die Aktion durchführen will.

Diese kann bei verschiedenen Auswahlmöglichkeiten erscheinen, besonders bei denen, welche Sicherheits- oder gravierende Änderungen an der Spiele-Konfiguration ermöglichen.



3.6 Charakterdesign

Da es sich bei unserer Anwendung um ein Spiel handelt stehen die Charaktere im Vordergrund und nicht die GUI als solches.

Daher zeigen wir Ihnen den aktuellen Status eines Charakters.

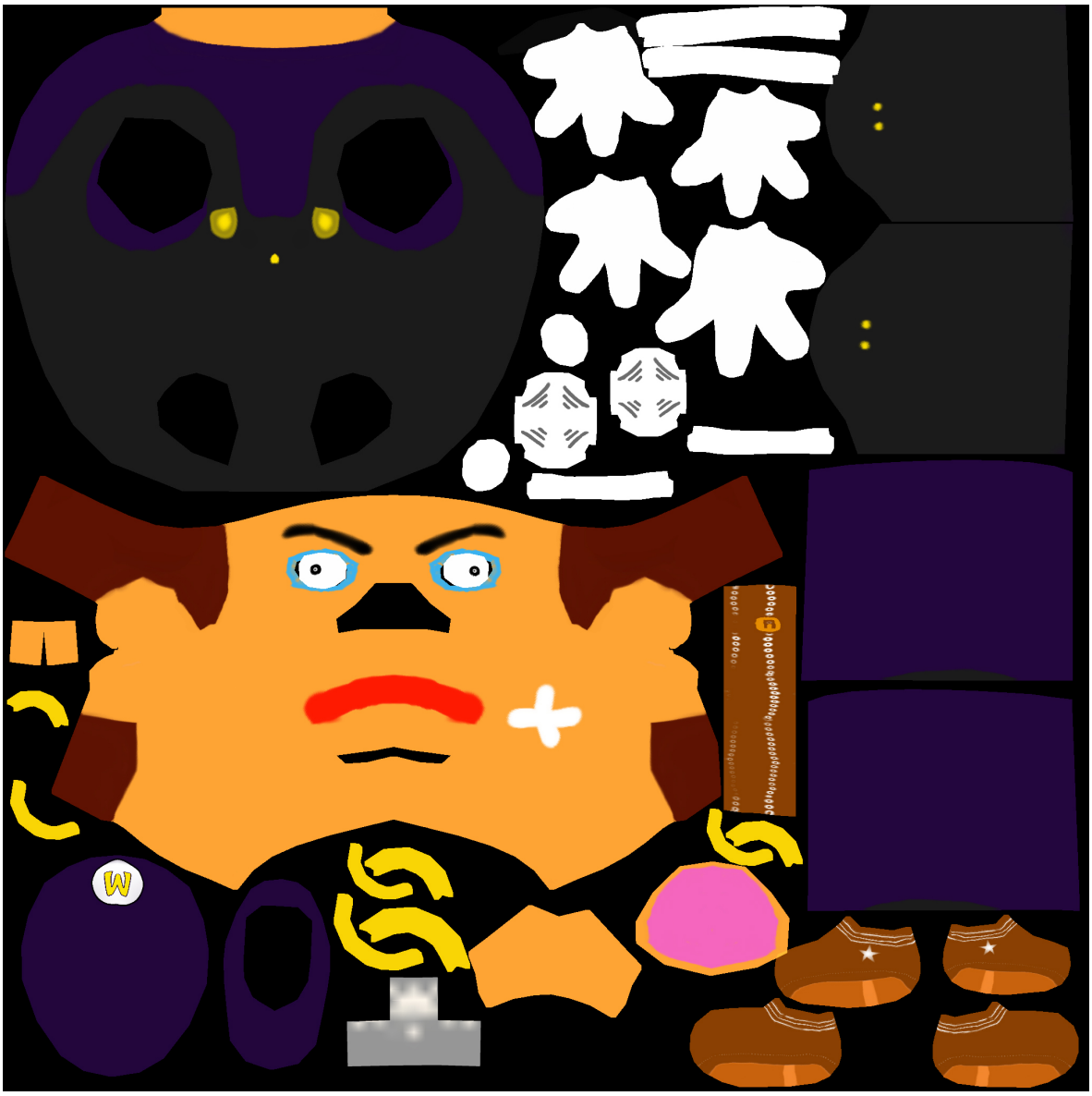
Dieser Charakter wird im Prototyp vorhanden sein. Polycount : 5,8k tris

In der Computergrafik werden Flächen bzw. Objekte ausschließlich in Dreiecke(triangles) dargestellt. Die Zahl des Polycounts gibt somit die Anzahl der Dreiecke in einem Mesh(3D Objekt) an.

In der derzeitigen Spielen werden Charaktere mit einem Polycount um ca. 12000 verwendet.

Da uns die notwendige Erfahrung in diesem Bereich fehlt, verwenden wir einen geringen Polycount mit der Option die Charaktere später zu erweitern.

Colormap für den Charakter:



Da es sich bei Mario World Wars um ein cartoonartiges Spiel handelt werden knallig Bunte Farben für die Charaktere verwendet.

Texturiert wird auf 2D Bildern. Das komplette Mesh wird somit vom 3D Raum auf einen 2D Raum(Textur) gemappt. Diesen Vorgang nennt man in der Computergrafik UV mapping.